Sophia: An Information Plane for Networked Systems

Mike Wawrzoniak, Larry Peterson, and Timothy Roscoe

IRB-TR-03-048

November, 2003

Intel Research Berkeley

Sophia: An Information Plane for Networked Systems

Mike Wawrzoniak and Larry Peterson
Princeton University

Timothy Roscoe
Intel Research Berkeley

Abstract

This paper motivates and describes an example network Information Plane, called Sophia, currently deployed on PlanetLab. Sophia is a distributed system that collects, stores, propagates, aggregates, and reacts to observations about the network's current conditions. Sophia's approach is novel: it can be viewed as a multi-user distributed expression evaluator in which sensors and actuators form the ground terms, and statements take on the complete expressiveness of a logic language like Prolog. This paper argues that this approach has several advantages in managing and controlling a complex, federated, and evolving network: (1) a declarative logic language provides a natural way to express the kinds of statements that are common to this application domain, through temporal and positional logic rules, facts and expressions; and (2) distributed evaluation of such logic expressions provides many opportunities for performance optimization yielding an efficient system.

1 Introduction

Consider a global overlay infrastructure like PlanetLab [5], designed to run on thousands of machines distributed worldwide, and host hundreds of network services that use and interact with each other and the Internet in complex and unpredictable ways. Managing this infrastructure—collecting, storing, propagating, aggregating, discovering, and reacting to observations about the system's current conditions—is one of the most difficult challenges such a networked system faces.

We propose a shared *Information Plane* to address this challenge. The Information Plane is a distributed system running throughout the network. It incorporates three functions: collecting information about network elements, evaluating statements (questions) about this state, and reacting according to conclusions drawn about the information. Building such an Information Plane is complicated by two factors. Firstly, information in the system is widely distributed in space (across all elements in the network), and constantly varies over time (we care about both state in the past and future events). Secondly, it is difficult to express meaningful statements about expected, anomalous, or desired behavior in the system: such statements are often highly complex, or can only be made at a very high level. Moreover, we cannot in general know what these statements are *a priori*, meaning that we must be able to formulate them at runtime.

In this paper, we describe an example network Information Plane we are building, called Sophia, that addresses these issues. Sophia models the three functions closely: a distributed set of *sensors* report data about aspects of the system, including both local state (e.g., the load or memory usage on a particular node) and the local perspective of the rest of the network (e.g., the reachability of other nodes). A *declarative programming environment*, also distributed, evaluates logic statements about the system. Finally, a set of *actuators* distributed throughout the network perform local actions (e.g. killing a misbehaving service on a node). Sophia is deployed today over PlanetLab.

Sophia's approach to decentralized management is novel: it can be viewed as a distributed expression evaluator in which sensors and actuators form the ground terms, and statements take on the complete expressiveness of a language like Prolog [3].

This has several advantages in managing and controlling a complex, federated, and evolving network. First, a declarative logic language provides a natural way to express the kinds of statements that are common to this application domain. Second, logic expressions can be transformed at runtime to a new form that is equivalent, but more suitable for efficient evaluation. Third, programs and data are equivalent in Sophia (they are Prolog-like predicates) making it just as easy to move the (sub)program to the data as to bring the data to the program. This makes Sophia efficient and easy to extend at runtime. It also allows the system to employ introspection to optimize its own performance, based on the information it is processing. We know of no other network management system with this property. Finally, since explicit notions of time and space are embedded into Sophia, it is easy to both transparently distribute expressions across the entire network, and to process both past and future information (i.e. process logs and schedule events).

The rest of this paper describes and motivates Sophia's architecture in detail, including the role of distributed unification to resolve logic expressions over a network, the use of capabilities to provide secure sharing of system data, and support for caching as both an optimization and a way to seamlessly incorporate logged information.

2 Design Considerations

This section identifies the requirements that influenced Sophia's design, and in the process, motivates the architec-

tural decisions we made. We also give illustrative examples that help make our case.

2.1 Expressivity

The first requirement is that the Information Plane must provide a language that makes it easy to express what we want—to make statements about the overall network state and behavior. This is harder than it sounds. One approach would be to use a domain-specific language whose design encodes our assumptions about possible states of the network. Alternatively, statements about the network might be represented within a predefined relational schema by means of a declarative query language.

We reject both of these approaches for the same reason: they incorporate *a priori* assumptions about possible states of the system. This characteristic is valuable in smaller and/or centralized systems since it provides useful abstractions and reduces the possibilities for bugs in code written in the language. However, in a wide-area, decentralized environment that evolves over time, such *a priori* assumptions will frequently become invalid. Users—those writing queries and specifying actions in the system—need a programmatic way to determine when such assumptions no longer hold, formulate new assumptions, and resolve apparent contradictions in system state.

Consequently, based on our experience with an InfoSpect [7] deployment on PlanetLab, we designed Sophia to use a logic programming language to express information about actual and desired system state. More precisely, Sophia supports both a high-level language designed for programmers (based on Prolog) and a low-level functor-based language (instruction set) that is used internally by Sophia. In general, alternative high-level languages (e.g., a functional language) could be used and mapped onto the low-level form, but for the purpose of this paper, we gloss over the distinction between these two levels and give examples using Prolog syntax.

The important point is that a computational model based on logical unification is at the heart of Sophia. The primary interface to the Sophia core is the `eval(`*Term*`)` functor, which uses unification to prove the *Term* to be true for some free variable substitution, or fails. Thus, when evaluated on a Sophia node, the expression:

```
eval( bandwidth(BwVar) ).
```

may succeed, returning the expression:

```
eval( bandwidth(81920) ).
```

Practically speaking, the evaluation returns the bandwidth value. In general, the *Term* can be a complete logic program. For example,

```
eval( (bandwidth(BwVar), BwVar < 65536) ).
```

will succeed with a value for `BwVar` if and only if it is within range. The actual values associated with many Sophia predicates are returned by sensors running on nodes throughout the network.

Unlike many situations where a logic programming model might be used, we are using it to express properties of a continuously-running networked system. This means we need to be able to express *where* and *when* logical predicates might hold. Our approach is to make both location and time an explicit part of every *Term* processed by Sophia, so that all expressions are bound to a certain environment and their interpretation in the system is precise.

All Sophia terms have a location component, meaning that the return of the evaluation of the previous example would actually be of the form :

```
eval( bandwidth( env(node(id42)), 81920) ).
```

meaning that the value of bandwidth at the particular node with id `id42` is 81920. If necessary, the relevant subexpression will be sent to an appropriate node for unification.

Similarly, because of the dynamic nature of the environment all facts and results may be time dependent, and so we bind all expressions with a *time* component. Thus, a more precise evaluation result of our ongoing example is :

```
eval( bandwidth( env(node(id42),
                     time(1057766930)),
                 81920) ).
```

Both `time` and `node` are part of an *environment* predicate associated with every functor. The predicate may contain variables and logic expressions describing them. As a consequence, it is easy to manipulate the environment component of expressions, allowing arbitrary time/location selection. As an illustration of the expressive power of this technique, when combined with a declarative language like Prolog, consider the following expression:

```
eval( (bandwidth( env(node(Node),
                      time(Time),
                      Time > 1057766930),
                  BwVar),
       BwVar > 65536) ).
```

which would, if possible, instantiate `Time`, `Node`, and `BwVar` variables to indicate a node where the bandwidth exceeded 65536 after time 105776693.

As a more comprehensive example, the following shows a rule declaration followed by its evaluation. The rule defines a `slice_bandwidth` expression to be the sum of slice bandwidths on all the nodes within the last 30 seconds. A slice is a PlanetLab distributed virtual machine abstraction, so evaluating this rule determines the total bandwidth consumed (`SliceBandwidth`) by a particular slice (`slice47`) across all PlanetLab nodes.

```
slice_bandwidth(Slice, Bandwidth) :-
      findall( [Node, BwVar],
               (node(Node),
                bandwidth(env(node(Node),
                              time(Time),
                              (Time>now-30)),
                          slice(Slice),
                          BwVar) ),
                Result),
      sumlist(Result, Bandwidth).
eval( slice_bandwidth( slice47,
                       SliceBandwidth) ).
```

2.2 Performance

Performance in an Information Plane has many facets. Raw sensor data must be obtained in a timely fashion, and distributed computation on the data must be performed efficiently in the face of network latency. Furthermore, some computations cannot be performed immediately since it refers to results in the future, and so must be deferred. Sophia addresses its performance demands with caching, evaluation scheduling, and planning.

2.2.1 Caching

Sophia nodes cache evaluation results including the facts extracted from sensors. This has several benefits. In situations where computation latency is more important than freshness of the data, no delay is incurred in evaluating a full expression and contacting the sensor. Since all Sophia expressions are bound to an environment that includes time, the freshness of the data can always be inferred if the desired time is not specified, and specified if required. For example, in the expression:

```
eval( bandwidth( env(node(id42),
                     time(SomeTime)),
                 BwVar) ).
```

the time is a free variable, and so a cached value of `BwVar` is obtained if one exists. On the other hand,

```
eval( bandwidth( env(node(id42),
                     time(SomeTime),
                     SomeTime >= now),
                 BwVar) ).
```

forces an up-to-date value to be obtained. Similarly, constraining `SomeTime` to a particular period in the past would effectively allow one to query a log of stored sensor readings. It should be clear from this example that Sophia's caching of partial results and sensor values, and the ability to specify evaluation times in the past, correspond to one and the same mechanism.

2.2.2 Scheduling

Evaluation scheduling provides the ability to pre-schedule expression evaluation ahead of time, so the results are available when and where they are needed. This is accomplished by evaluating expressions with a time value in the future. Using this mechanism, it is possible to schedule expressions to be always evaluated within a certain interval, thereby refreshing the cache with fresh values.

2.2.3 Planning

In database systems, *query planning* or optimization is the stage of processing that takes a purely declarative description of the query results and generates the set of operations to obtain them efficiently. In Sophia, the analogous operation is *evaluation planning*, and includes notions of location and time as well as translating an expression into an efficient form for evaluation. For example, to achieve good performance it may be optimal to evaluate a subexpression at a location in the network close to a dependency. Similarly, it may be best to schedule an evaluation only at the right time so that the dependencies are already resolved. Finally, it may be beneficial to rewrite the original expression into a different logically equivalent expression of a form which can easily be decomposed into components that can benefit from separate planning.

The example expressions we have shown so far have not addressed the issues of what to evaluate and where; however Sophia's real power comes when this information is derived by the system from a higher-level specification.

In Sophia, planning is implemented at the same level as query evaluation and proceeds by unification: the planner is a set of logical rules that expand high-level terms into the kinds of explicit evaluations we have just seen. This lack of an explicit boundary between query specification and query planning/optimization allows great flexibility. It is relatively straightforward to include static query-planning techniques analogous to those found in databases, where a purely declarative query is translated into a plan as a separate step. However, with Sophia we can potentially do much better: query plans can be formulated on the fly as the evaluation proceeds, using partial results to re-optimize the query.

This can be extremely powerful in a distributed environment. Although deriving an efficient plan is a difficult task, the important point is that this performance optimization is possible because of Sophia's logic model, together with the integration of time and location at a low level. We view this as an exciting area for future work.

2.3 Failures

The Information Plane must function in a large-scale, widely distributed environment in which failures and anomalies must be treated as the norm, not an exception. These conditions can make certain expression evaluations impossible, since it may not be feasible to satisfy some of the dependencies at any given point in time.

We recognize that it is often better to get *an* answer than to not get the most *complete* answer. As a consequence, Sophia's expression evaluation mechanism deals with partial failures using logic *holes*—a sub-expression that cannot

be evaluated, and hence, analogous to a conditional proof and partial result processing in relational databases [6]. In a sense, this allows Sophia to sacrifice completeness for performance. The expression is not fully evaluated until all logic holes are evaluated, however, the partial results may be useful. Moreover, this allows for incremental evaluation by proving the holes as the dependencies are satisfied. Said another way, decomposing an expression so as to isolate holes is a critical part of the evaluation planning problem.

2.4 Extensibility

Finally, the Information Plane must be extensible. This means it must be easy for any one user to add new functionality over time, but also that many different users must be able to query the Information Plane in different ways. Our decision to base Sophia on a programming language is certainly an important part of the answer, but in addition, Sophia has been engineered to require a minimal core on each node that is extended with loadable modules to provide a complete system. The loadable modules are sets of rules, which are asserted into Sophia's term database. The real challenge is how to manage the various contexts in which expressions are evaluated in a way that supports the desired levels of isolation versus sharing.

Our approach is to run a single Sophia system with multiple modules loaded into it. Sophia uses capabilities to protect modules, grant and revoke privileges, and support module composition. Capabilities in Sophia take the form of simple rule declarations and are transparently handled by the low-level language. We have built a "capabilitizing compiler" that permits the high-level language to hide capabilities from the programmer. The following discusses the various roles that capabilities paly in more detail.

2.4.1 Privileges

Capabilities are used to assign and enforce system privileges. For example, if only some users are allowed to evaluate the previously described `bandwidth` rule, the rule is turned into a capability at the functor level. That is, instead of

```
bandwidth(Val) :- read sensor.
```

the rule is changed to

```
cap389456(Val) :- read sensor.
```

and to evaluate the `bandwidth` rule, the user must have the right capability, which is also just a rule

```
bandwidth(BwVal) :- cap389456(BwVal).
```

Note that capabilities are based on randomized 128-bit strings, although we shorten them in the examples for the sake of convenience. Also note that the local unification algorithm restricts the user evaluations from listing the contents of the term database. Otherwise, it would be possible to list all the capabilities and make them useless.

Since capabilities are just rules, many elaborate protection systems can be easily implemented. For example, to make the `bandwidth` capability revocable, the privilege granter creates an intermediate capability to the real capability, e.g.,

```
cap568907(Val) :- read sensor.
cap389456(Val) :- cap568907(Val).
```

where `cap389456` is an intermediate capability that currently references the target capability `cap568907`. The user is given only the intermediate capability. To revoke a user's privilege of reading the sensor, the privilege granter simply removes the target capability, so the user's evaluation of `bandwidth` rule always fails since evaluation of the intermediate capability fails. Similarly, an *use-N-times* capability could delete itself during its N-th evaluation, a two-way authorization capability could check additional arguments for keys, and so on.

2.4.2 Module Protection

In addition to granting privileges, modules are protected from each other using capabilities. If a module is to be completely private—that is, all of its predicates are to be visible within the module only—all of its predicates are transformed into capabilities. The capabilities are then grouped to a key-capability rule, and the user of the private module must possess that rule to unlock the module for evaluation.

2.4.3 Module Composition

Capabilities are also the key mechanism used to control module composition. All rules of a module are by default transformed to private capabilities visible only within the module. However if a module wants to expose an interface, it may explicitly leave a rule with a well-known name declared without turning it into a capability. Any other module would be able to evaluate the public rule. On the other hand, if a module wishes to interface only with certain modules, it may transform the given rule into a capability, and then pass the capability to only the user that is allowed to use the interface. A more elaborate capability rule could be composed to authenticate the user.

2.4.4 Capabilities and Caching

It is important to notice how well the capability rules and caching interplay with each other. One danger of creating interfaces to common predicates through capabilities is that caching could be performed on the capability rules, causing the equivalent rule evaluations through different interfaces to miss the cache. On the other hand, exposing the cache to the untrusted entity might jeopardize its module protection. In Sophia however, the evaluation of capability rules that provide access to the same target rule share the cache without compromising protection. The results of the target rule are cached and cache hits are on the target rule,

thereby avoiding redundant evaluations. At the same, time users gain no unauthorized access to the shared cache.

3 Implementation

This section outlines the main components of the implementation of Sophia which currently runs as a distributed service over PlanetLab. Users connect to any Sophia node and evaluate system-wide expressions, assert facts, or load modules.

Each node in the system runs a minimal local core, with most of Sophia's functionality implemented as loadable modules. This allows for alternative implementations of the system components to be available on a running system. User modules can choose to compose with the preferred implementation of the system modules, or user modules can implement specialized system modules (given they have the necessary privileges). The local core running on each node consists of the following five components.

First, Sophia uses a single, flat *logic terms database* for storing all of its terms. The database supports `assert Term` and `retract Term` operations, and is used by the unification engine to match terms. The stored terms include all predicate rules and facts, as well as loadable modules, which are just a set of terms. For example, the database may contain rule terms of a slice policing module, some of which are:

```
slice_malicious(Slice) :-
        slice_bandwidth(Slice, Bandwidth),
        Bandwidth > bwlimit.
slice_malicious(Slice) :-
        slice_ip_dest(Slice, DstCount),
        DstCount > ip_dest_limit.
slice_police(Slice) :-
        slice_malicious(Slice),
        slice_stop(Slice).
slice_police_all :-
     findall( [Slice],
              ( slice(Slice),
                slice_police(Slice) ), _ ).
```

In reality, all the terms in the database are stored in the low-level functor-based format, constrained with the environment predicate (i.e., time and location are bound in every term), as follows

```
( FunctorName, Env, Term1, ..., TermN )
  Env = env( time(Time), node(Node), _)
```

Second, Sophia includes a *local unification engine* that is based on standard logic unification, with the following differences. First, environment (time and location) dependent caching is a part of the engine's mechanism, favoring matches in the local cache rather than delegating evaluation to another node. Second, to protect users from listing available capabilities, there is a restriction on rules listing other terms in the terms database. Third, the term database is augmented with session-dependent terms for the sake of unification. A session-manager maintains these private terms, which typically include the user's capability rules.

Third, Sophia interfaces with sensors and actuators of the host system [8], which effectively serve as I/O for Sophia. The interface provides a bridge to access sensory data from the host environment, as well as act on its actuators. The interface exposes the sensors and actuators as regular terms, which makes them transparent to the unification process. For example, in support of the slice policing module introduced above, a sensor showing IP destinations contacted by the host and an actuator to stop a PlanetLab slice are exposed as transparent terms:

```
ip_dest(DestList).
slice_stop(Slice).
```

Fourth, a *remote evaluator* is responsible for delegating evaluation of an expression to a particular remote Sophia node. It handles the networking and Sophia protocol between the nodes. The results of the evaluation are cached localy for local unification. For example, the evaluation

```
eval( env(node(id82)),
      ip_dest( env(node(id82),
                   time(T) ),
      Dest) )
```

is delegated to node id82 unless there is a local cache hit. It is interesting to note that `eval` is just a predicate and can be statically specified as part of rule definitions, allowing static specification of an evaluation plan. Note that any strategy for distributed unification is implemented in a separate module; it is not part of the remote evaluator.

Finally, an *expression scheduling mechanism* is responsible for maintaining the calendar of evaluations scheduled for the future. It inserts appropriate event handlers into Sophia's event facility. A utility module provides a predicate that allows for scheduling expressions to be evaluated with a certain frequency, keeping their results fresh. For example,

```
eval( schedule( env(time(T), T>=now),
                fresh(10),
                slice_police_all) ).
```

schedules the rule `slice_police_all` to be re-evaluated every 10 seconds.

4 Related Work

The management of large networked systems is an entire field in itself. Commercial network management systems such as HP OpenView and Micromuse Netcool provide simplified interfaces to routing functionality such as topology discovery, service provisioning, and equipment status checks, and are routinely used by ISPs. Host-oriented management systems such as IBM's Tivoli and Computer Associates' UniCenter address the corresponding problems of

managing large numbers of desktop and server machines in an enterprise. Both kinds of system are aimed at single organizations with well-defined applications and goals seeking to manage and control the equipment they own. Managing a wide-area, evolving, federated system like PlanetLab (or the Internet as a whole) poses different challenges, however, and a goal of Sophia is to better understand this new problem space.

The case for Sophia as a framework in which the behavior of a widely-distributed network can be monitored and controlled is similar to that of the Knowledge Plane [2]. In fact, Sophia's high-level model of using sensors to collect information, a distributed expression evaluator to ask questions and make statements about that information, and a set of actuators to take actions in response to conclusions drawn about the information, evolved out of early discussions about the architecture of the Knowledge Plane. In other words, one could view Sophia as an incarnation of the Knowledge Plane for PlanetLab, although we also note that PlanetLab provides multiple vantage points from which Sophia could be extended to serve as a prototype Knowledge Plane for the Internet as a whole.[1]

Sophia also owes much of its design philosophy to InfoSpect [7], which argued that a declarative programming language like Prolog is a natural choice for expressing complex queries about system behavior in a network monitoring system. The main way in which Sophia extends InfoSpect is that it is distributed over the network. At a more detailed level, Sophia also has explicit notions of time, space, caching, modularity, and capabilities, whereas InfoSpect is in essence an evolving but centralized set of Prolog queries about instantaneous, remotely determined system state. We are currently re-deploying InfoSpect as an application on top of in Sophia.

It is instructive to compare Sophia to distributed query processing engines currently being deployed in wide-area networks, such as PIER [4] (which applies a relational model to wide-area queries) and IRIS [1] (which has a hierarchical data model using XML and XPath). Although these two systems do not have the explicit goal of monitoring and managing the network (they draw their motivation from distributed databases and sensor networks, respectively), they are both being used alongside Sophia to monitor PlanetLab, and in fact, we defined a universal sensor interface for PlanetLab [8] so that all three systems (as well as others) could have access to all information we collected about the network.

These three systems represent different design points in terms of representation and expressiveness: Sophia is designed around a declarative logic programming model where the location of code execution is both explicit in the language and can be calculated in the course of the evaluation. In contrast, the details of execution of queries in PIER is left to the underlying implementation to optimize (in the tradition of relation databases), subject to the constraints introduced by the use of a DHT for rehashing. In IrisNet, the hierarchical structure enforces where query execution must occur, and consequently efficient execution is a question of establishing an appropriate hierarchy when the system is deployed. The consequence is that Sophia queries can potentially be much more sophisticated in both expression and optimization—both the user and the system can participate in the evaluation planning to varying degrees.

[1] We use the term "Information Plane" rather than "Knowledge Plane" because we do not currently explore the use of learning algorithms, and other AI techniques, to adapt to unforeseen behavior.

5 Conclusions

Sophia represents a promising new direction in managing widely-distributed and continuously-changing network systems. Our experience to-date suggests that Sophia's foundation in a computational model based on logical unification is particularly powerful: (1) it defines a natural way to express assumptions and requirements; (2) it provides a rich space for exploring adaptive distributed query planning algorithms, including algorithms that can work around transient failures; and (3) it enables a flexible tradeoff between privacy and sharing through the use of capabilities. However, much work remains to be done to fulfill the potential of these opportunities. Most importantly, we need to learn how to use (program) tools like Sophia to help us better manage and control the increasingly complex networks we are deploying.

References

[1] IrisNet: Internet-Scale Resource-Intensive Sensors. `http://www.intel-iris.net/`, July 2003.

[2] D. Clark, C. Partridge, C. Ramming, and J. Wroclawski. A Knowledge Plane for the Internet. In *Proc. ACM SIGCOMM, Karlsruhe, Germany*, August 2003.

[3] W. Clocksin and C. Mellish. *Programming in Prolog*. Springer Verlag, 1984.

[4] R. Huebsch, J. M. Hellerstein, N. Lanham, B. T. Loo, S. Shenker, and I. Stoica. Querying the Internet with PIER. In *Proc. VLDB*, May 2003.

[5] L. Peterson, D. Culler, T. Anderson, and T. Roscoe. A Blueprint for Introducing Disruptive Technology into the Internet. In *Proc. HotNets-I, Princeton, NJ, USA*, October 2002.

[6] V. Raman and J. M. Hellerstein. Partial Results for Online Query Processing. In *Proc. ACM SIGMOD, Madison, WI, USA*, June 2002.

[7] T. Roscoe, R. Mortier, P. Jardetzky, and S. Hand. InfoSpect: Using a Logic Language for System Health Monitoring in Distributed Systems. In *Proc. of the ACM SIGOPS European Workshop, Saint-Emilion, France*, September 2002.

[8] T. Roscoe, L. Peterson, S. Karlin, and M. Wawrzoniak. A Simple Common Sensor Interface for PlanetLab. PlanetLab Design Node PDN-03-010, `http://www.planet-lab.org/pdn/pdn03-010.pdf`, March 2003.